A Benchmark-based Performance Model for Memory-bound HPC Applications

Bertrand Putigny, Brice Goglin, Denis Barthou

A Benchmark-based Performance Model for Memory-bound HPC Applications

Bertrand Putigny*†, Brice Goglin*†, Denis Barthou*†‡
*INRIA Bordeaux Sud-Ouest, †LaBRI, ‡Polytechnic Institute of Bordeaux
bertrand.putigny@inria.fr, brice.goglin@inria.fr, denis.barthou@inria.fr

***Abstract*—The increasing computation capability of servers comes with a dramatic increase of their complexity through many cores, multiple levels of caches and NUMA architectures. Exploiting the computing power is increasingly harder and programmers need ways to understand the performance behavior.**

We present an innovative approach for predicting the performance of memory-bound multi-threaded applications. It relies on micro-benchmarks and a compositional model, combining measures of micro-benchmarks in order to model larger codes. Our memory model takes into account cache sizes and cache coherence protocols, having a large impact on performance of multi-threaded codes. Applying this model to real world HPC kernels shows that it can predict their performance with good accuracy, helping taking optimization decisions to increase application's performance.

***Keywords*—memory model; timing prediction; micro-benchmarks; caches; multicore.**

I. Introduction

High performance computing requires proper software tuning to better exploit the hardware abilities. The increasing complexity of the hardware leads to a growing need to understand and to model its behavior so as to optimize applications for performance. While the gap between memory and processor performance keeps growing, complex memory designs are introduced in order to hide the memory latency. Modern multi-core processors feature deep memory hierarchies with multiple levels of caches, with one or more levels shared between cores.

Performance models are essential because they provide feed-back to programmers and tools, and give a way to debug, understand and predict the behavior and performance of applications. Coherence protocols, prefetchers, replacement policies are key automatic hardware mechanisms that improve peak performance, but they also make parallel architecture modeling much harder. Besides the detailed algorithms used by these mechanisms, the latencies involved or the size of the buffers implied are not precisely quantified by hardware manufacturers in general. Finally, even if the hardware resource organization may be known at runtime thanks to tools such as hwloc [1], the way they are actually used depends on the application, namely how it allocates, reuses and shares buffers between the computing tasks.

These difficulties hinder precise performance modeling, in particular for memory performance. Many previous works have abstracted away the cache behavior by modeling essentially compulsory, capacity and conflict misses (the 3C). The resulting analytical models, through the computation of stack distances, capture data locality and offer a guidance for compiler optimizations (see [2], [3], [4] for instance). Most of these previous works target the cache behavior of single-threaded codes. Some recent works focus on how to model cache misses [4] for multi-threaded codes, or on how to model cache coherence traffic [5]. However, they do not consider simultaneously the impact of cache misses, of coherence and contention coming from shared caches. A 5C model, accounting for compulsory, capacity, conflict misses, contention and coherence remains to be found in order to help study the impact of factors such as the data set size for each thread or the code scalability on multi-cores.

In this paper we present a novel performance model that allows detailed performance prediction for memory-bound multi-threaded codes. This differs from the previous approaches by predicting the cumulated latency required to perform the data accesses of a multi-threaded code. The model resorts to micro-benchmarks in order to take into account the effects of the hierarchy of any number of caches, compulsory misses, capacity misses (to some extent), coherence traffic and contention. Micro-benchmarks offer the advantage of making the approach easy to calibrate on a new platform and able to take into account complex mechanisms. In addition these benchmarks can be used as hardware test-beds, *e.g.* to choose between several architectures which one best suits the needs.

This paper is organized as follows: first, Section II presents the scope and an overview of our work. Our model is later detailed in Section III as a combination of hardware and software models. Then Section IV presents the benchmarks used to build the hardware models. Finally Section V details the use of our model to predict real world code performance.

II. Scope and Model's Overview

Our model takes as input the source code to analyze. The code can be structured with any number of loops and statements, and we assume the data structures accessed are only scalars and arrays. Parallelism is assumed to be expressed in OpenMP parallel loops. The iteration count of the loops have to be known at the time of the analysis, and the array indexes have to be affine functions of surrounding loop counters and of thread ids.

Predicting the time to access memory usually requires to build a full theoretical performance model of the memory hierarchy. This work is however difficult due to the complexity of all the hardware mechanisms involved in modern architectures. Instead we choose to build a memory model upon benchmarks that are used to capture the hardware behavior of common memory access patterns while hiding the impact and complexity of these mechanisms inside the benchmark output.

The main difficulty with this approach is to find a set of benchmarks that characterizes hardware precisely, and to keep this set as small as possible. Then we try to rebuild the application memory access pattern by combining the outputs of these basic benchmarks. We found that in cache coherent architectures, the state of the target cache lines has a large impact on performance. Concurrent accesses to shared data buffers lead to cache-line bounces between cores due to the need to maintain coherence between the existing data copies in their caches. Thus we build a set of benchmarks that gives us insight about the read and write latency to cache lines for every of the state of the MESI protocol [6]. Indeed most cache coherent processors use protocols that are based upon MESI.

In order to predict the time needed to access memory for a given application, we decompose it into a *memory access pattern*. This pattern tells us the amount of memory access and how it is accessed (*i.e.* reading or writing). If we suppose that we know the full state of memory (*i.e.* the cached addresses and their locations), we can *i)* reconstruct the state of memory after every access, and *ii)* read each access duration from the output of our benchmarks. By taking memory events one after another we can track the state of data in caches and construct a formula that will predict the time needed to access memory for the whole application.

III. Program and Memory Models

In Section III-A we present how programs are represented in order to be able to apply the prediction. Then we detail the view of memory used in our work in Section III-B. Finally Section III-C describes how the model is used for predicting execution times.

A. Program Model

Input codes are OpenMP parallel codes, using only parallel for loop constructs. Our model represents programs by only considering their memory access patterns since our work focuses on memory-bound codes. OpenMP codes are transformed into a simplified code, where statements are memory accesses, each statement accessing a memory region.

Read/write array accesses in OpenMP parallel for loops are transformed into `read` or `write` statements accessing all the elements read or written by a thread. The parallel for loop is then transformed into a parallel OpenMP region. For instance, the following DAXPY computation with n threads, assuming N is a multiple of n:

```
double X[N],Y[N];
#pragma omp parallel for
for (i=0; i<N; i++)
    Y[i] = a * X[i] + Y[i];
```

is modeled as the following code:

```
double X[N],Y[N];
#pragma omp parallel
    int t = omp_get_thread_num();
    int T = omp_get_num_threads();
    read ([t*N/T:(t+1)*N/T-1:8],X);
    read ([t*N/T:(t+1)*N/T-1:8],Y);
    write([t*N/T:(t+1)*N/T-1:8],Y);
```

where $[t * N/T : (t + 1) * N/T - 1 : 8]$ denotes the array index region accessed by a thread, with the three values corresponding to the lower, upper indices, and the stride (8 for double) of the region. This notation corresponds to the triplet notation used in compilers. The data region $X[t*N/T : (t+1)*N/T-1 : 8]$ is defined as a data *chunk* in the following. The array index region is a function f of N, T and t. When this is clear from the context, we will denote this region with $f(t)$ (assuming the size and total number of threads are constants). This function f is defined as the *mapping function* in the following.

In this representation, the sequence of reads and writes, where initially two reads alternates with one write, has been replaced by two streams of reads followed by a stream of writes. In order to represent the capacity of modern architectures to load and store multiple elements per cycle, we consider an execution model where `read` and `write` statements can be dispatched to parallel units. For instance, on an architecture able to perform one read and one write at a time, the two read accesses can only be performed in sequence, while the write access can be performed concurrently to the reads. This corresponds to the asymptotic optimal use of the hardware units, that can be reached through code optimization.

Similarly, element-wise read/write accesses in sequential loops are subsumed into array region accesses, using `read` and `write` statements. For instance, the code on the left hand side is modeled as the code on the right hand side.

```
for (i=0; i<N; i++)                  -> write([0:N-1:8],X);
    X[i] = k;                           read ([0:N-1:8],Y);
    k = Y[i];
```

In the original code, the first statement depends on the second statement of the previous iteration. This loop-carried dependence is not represented in our model. In the execution model proposed above, both read and write array region statements will be executed simultaneously. That represents the fact that on modern hardware, an optimized version of the original code is still able to execute one load (`X[i+1]`) and one store (`Y[i]`) per cycle.

In the previous code there is only one data chunk for X, while in the parallel DAXPY code, the same region

corresponds to N/T data chunks. In our memory model (presented in the following section), we keep track of the state of each data chunk in the coherency protocol. To do so, the data chunks accessed by a program have to be considered as atomic pieces of data, in particular, two different chunks cannot share array elements, as for array X and Y in the previous examples. This can be obtained by decomposing `read` and `write` statements into statements accessing smaller data chunks, so that the chunks represent a partition of the array elements. For instance, in order to manipulate the same data chunks in the previous code and in the parallel DAXPY, the two `read` and `write` statements in the previous code are decomposed into N/T statements:

```
for (t=0; t<T; t++)
    write([t*N/T:(t+1)*N/T-1:8],X);
for (t=0; t<T; t++)
    read ([t*N/T:(t+1)*N/T-1:8],Y);
```

where T denotes the number of threads used by the program. This assumes that the number of threads is constant for the execution of the program. Splitting array regions can lead in the extreme case to one region for one array element. We assume this operation does not create a number of region dependent on the size of the array.

This representation only handles exactly array index regions that can be represented by triplets. For OpenMP, this is limited to parallel loops with static scheduling, `MASTER` sections (only thread 0) or `SINGLE` (can be modeled with thread 0). For other cases, or for triangular sequential loops, the chunks are then over-approximated.

B. Memory Model

The objective of the memory model is to predict the time required to perform the access statements, taking into account the impact of the coherency protocol. To this end, we associate a state in the coherency protocol to chunks and tabulate the time to perform an access statement to a chunk of any given size, with any number of threads and in any state. As the impact of the cache hierarchy is taken into account by a latency function, the memory hierarchy is entirely modeled as one level of

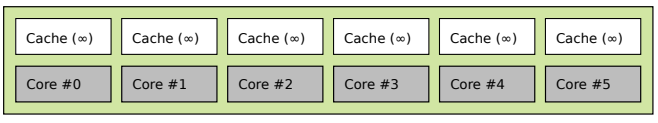

Figure 1: The memory hierarchy is modeled as one level coherent, private and infinite capacity caches.

coherent, private and infinite capacity caches, with one cache per core, as depicted in Figure 1. We will assume in the following that threads are pinned to cores. A thread has therefore a corresponding cache. We define a latency function, giving the time to access a chunk of data as a function depending: on the size of the chunk and on its stride, on the state of this data in the caches, and on the number of threads that access data simultaneously.

The state of a chunk consists in one of the MESI states and on the list of threads that have the chunk in their own cache. Adapting this technique to variants of the MESI protocol, such as MESIF or MOESI, is straightforward. These chunk states can be:

$M_{\{t\}}$	This data chunk is in state Modified in the cache of the thread t.
$E_{\{t\}}$	This chunk is only in the cache of the t^{th} thread, in Exclusive state.
S_Ω	The chunk is in Shared state for all threads in Ω.
I	This chunk is not present in any cache. At the beginning of the program, all chunks are in the Invalid state.

Therefore, for any array X and any mapping function f associated to X, the state of a chunk $X[f(t,N,T)]$, for any t, N, T, is updated according to the accesses to it. This state will be denoted $X[f(t,N,T)].state$.

The transitions between these states correspond to the actions performed on the chunks. A chunk can be either read by a set of threads $t \in \mathcal{T}$ (action denoted $\mathbf{L}_\mathcal{T}$ for load) or written by one thread t (action denoted $\mathbf{S}_t$ for store). Figure 2 defines the transition function δ for these states, according to the type of access and the ids of the threads accessing the chunk. For instance, consider a chunk in state Exclusive, in the cache of thread 0. This state is denoted $(\mathbf{E}, \{0\})$. Writing this chunk with thread 1 changes its state to $\delta((\mathbf{E}, \{0\}), \mathbf{S}_1) = (\mathbf{M}, \{1\})$. Each chunk maintains its own state and each state keeps track of the threads ids that have a valid copy of the chunk in their cache.

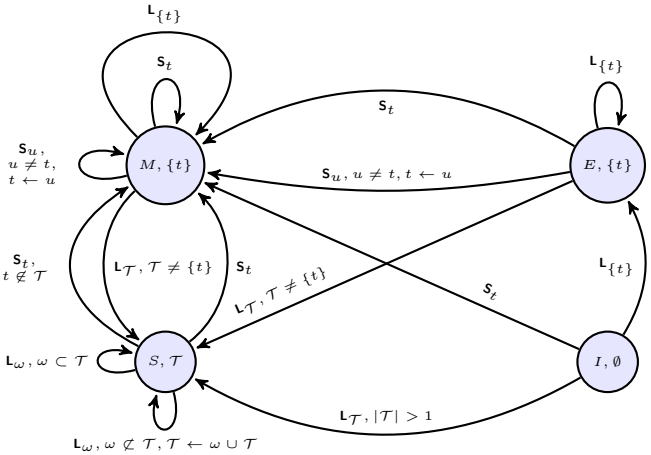

Figure 2: Automaton for tracking chunk's states and for selecting the benchmarks that represent each memory access step.

At the beginning of the program, the data chunks are in the state $(I, \emptyset)$. Executing read and write statement in our program model changes chunk states and the latencies for these transitions are defined by micro-benchmarks.

C. Time Prediction

The prediction of the memory access time of an OpenMP parallel code relies on two steps: The time measurement of all transitions in Automaton 2, for all data sizes and thread numbers, and the definition of a semantics connecting the `read` and `write` statements with the transitions within this automaton.

All transitions of the Automaton 2 are associated to timing functions obtained through micro-benchmarks. These transitions depend on the type of access (read/write), and of the number of threads performing this same transition. Indeed, some read transitions $\mathbf{L}_{\mathcal{T}}$ can be taken by threads in a subset $\mathcal{T}$ of threads. The time to execute a transition depends additionally of the initial state of the chunk, its size and stride, and of all transitions taken by other threads concurrently (sharing bandwidth to memory and last level caches). In order to limit the number of micro-benchmarks, the timing function considered for a transition will only depend on the initial state of the chunk, its size and stride, the total number of threads, assuming they are all performing the same transition, on a different chunk with same characteristics. These benchmarks are thus taking correctly into account capacity misses and limited bandwidth when threads access simultaneously different chunks. However, the latency of a parallel access to the same data chunk is not considered and will be approximated by a parallel access to multiple chunks of the same size. Considering a data chunk c read by thread t in a parallel region with T threads, the time to perform this read will be denoted $\mathbf{L}(c.state, c.stride, c.size, T)$. The description of how the benchmarks are built is detailed in the following section.

The timing functions provided by the benchmarks are then used to build a timing function for the program. The states are computed after each `read` and `write` statement. To aggregate these timing functions, we consider separately sequential phases in the program from parallel phases (within OpenMP parallel loops). Besides, two considerations are taken into account:

- At least one read and one write memory access can be executed in parallel on modern architectures. For each phase, the total duration for a sequence of memory access statements is computed as the maximum of the aggregated time to perform the reads, and of the aggregated time to perform the writes.
- A sequence of read/write chunk statements can represent element-wise accesses that are interleaved. We consider therefore the size of the data accessed during a read/write chunk statement as the total size of all chunks accessed during this phase.

Now, the aggregation time for reads (or writes) in sequential phases of the program simply corresponds to the sum of the individual time for read/write statements.

Algorithm 1 shows how the time is computed and states are updated. **L** and **S** are the functions obtained from the micro-benchmarks. To take into account the interleaving of accesses, the total working set size for each thread is first computed (line 1). Read/write accesses are assumed to be on data of this size (parameter $s(t)$ in **L,S** lines 7, 9), and the time is scaled down to the actual size of the chunk. States are then updated according to the type of access.

Algorithm 1: Compute the time θ of read/write statements in a parallel region.

```
   Input: S_1 ... S_n // Sequence of statements in a parallel region
   Input: L, S // Functions defining transition timings
1  forall the t = 1..T do // Compute working set of thread t
2      s(t) ← Σ_{i=1}^{n} chunk(S_i, t).size
3  for i = 1..n do
4      forall the t = 1..T do
5          c ← chunk(S_i, t) // Chunk accessed by thread t, S_i
6          if S_i is read then
7              θ_L(t) += L(c.state, c.stride, s(t), T) * c.size/s(t)
8              c.state ← δ(c.state, L_t)
9          else
10             θ_R(t) += S(c.state, c.stride, s(t), T) * c.size/s(t)
11             c.state ← δ(c.state, S_t)
12 θ = max_{t=1..T}(θ_L(t), θ_R(t))
```

Read (and write) statements can be executed in parallel. As all chunks may not be in the same state, these memory accesses may not take the same time for all chunks. Zhang *et al.* [7] have shown that the sharing pattern of threads on multi-threaded application are often very regular: in this case all chunks accessed within a read or write in a parallel phase are in the same state. In order to ensure prediction even when different chunks are in different states, the latency for all chunk accesses is simply assumed to be the longest one among all accesses (max in line 12).

IV. Benchmarking Framework

This section presents the benchmarking framework we developed[8]. In the following, memory latencies are presented as bandwidths. These metrics are equivalent since the data size accessed is known but bandwidth makes performance comparison easier on graphical plots.

A. Benchmark Design

We build a set of elementary benchmarks to measure the performance of each memory level (*i.e.* cache levels and main memory). In order to achieve near peak memory performance, these codes are in assembly and perform read or write access to a buffer of a given size. These building blocks are used to time each of the 15 transitions of automaton 2. To do so, these benchmarks first set a buffer into the initial state of the automaton, then access the buffer (by a read or write) and record the bandwidth achieved. Moreover, another set of benchmarks evaluates the possible slowdown due to contention when the transitions are taken in parallel by several sets of threads. Our approach differs significantly from other tools such as the Stream benchmark [9] because it takes coherence and possible contention into account.

The performance of benchmarks involving either the state $(S, \mathcal{T})$ or a transition $\mathbf{L}_{\mathcal{T}}$ depends on the number of caches holding the same chunk of memory or the number of threads accessing this data, hence it depends on their location in the machine. For instance, to execute the transition from state S to state M with a store requires to use at least 2 threads on a 8-core socket, hence there are 28 possible configurations. The same benchmark with 3 threads in parallel theoretically requires 56 tests. In order to avoid the combinatorial explosion of running all these configurations for every benchmark, only one configuration is tested by benchmarks for a given number of threads. This leads to n different runs total instead of to 2^{n-1} if we ran all combinations for all amounts of threads. Fortunately the performance of a benchmark with a given number of threads does not depend on their location within the socket since the inner-socket architecture of our target platform is flat.

B. Benchmark Outputs

All results in the remaining sections of the paper are obtained on a dual-socket 8-core 2 GHz Xeon E5-2650 host based on the Intel Sandy Bridge architecture.

Figure 3 presents the timing for the transitions corresponding to the same action, a load $\mathbf{L}_{\{u\}}$ when u is not among the threads of the initial state (either M, E or S). This transition hence corresponds to a miss, with possible coherency messages. Note that in this case, counting the number of cache hits and misses, as commonly done [10], is not sufficient to predict performance. Indeed, for these three configurations, the numbers of hits and misses are exactly the same, while the effective latencies are very different.

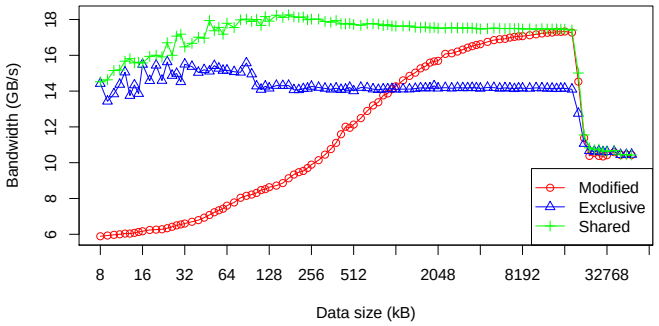

Figure 3: Load Miss benchmarks.

Latency highly depends here on the initial coherency state of the data. There is almost a factor 2 difference between a load to modified data (6-8 GB/s for up to 64KB of data) w.r.t. a load to shared data (14-18 GB/s for the same range), when this data fits in L1. The reason is that making a read request for data in the state M in another L1 cache requires first to invalidate it in the L1 (and possibly L2). When data is large enough, it moves during the write from L1 to the L3. The read then shows L3 performance (around 18GB/s for sizes larger than 8MB).

To assess the effects of contention occurring when the same caches are accessed concurrently simultaneously, we generate benchmarks where all threads run the same code, on different data. Threads do not share data, only bandwidth and the capacity of the last level cache. For a given transition, we analyze the performance degradation defined as the ratio between the bandwidth measured for one benchmark over the bandwidth measured when n similar benchmarks are running in parallel. A ratio of 1 means that there is no contention, each thread can use the same bandwidth as it would if it was running alone on the machine. A ratio greater than 1 is the factor dividing the available bandwidth per thread.

This ratio does not necessarily represent contention within caches or on the memory bus. We actually observed no cache contention on the Intel Sandy Bridge micro-architecture used for our tests, while the AMD Bulldozer micro-architecture shows some. The performance loss can also result from the limited capacity of the shared L3, observed when threads saturate it. They cause some *parallel capacity misses*, which looks like cache contention on the benchmark outputs.

Figure 4 shows the bandwidth degradation ratio when a store hit occurs.

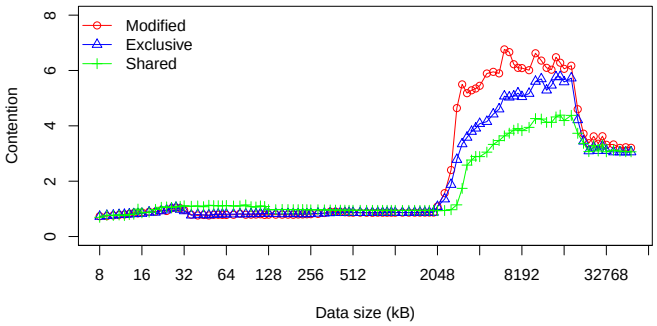

Figure 4: Store Hit bandwidth degradation for 8 threads.

As shown on Figure 4 there is almost no contention on private caches: Every independent cache can deliver the same bandwidth when it is accessed alone or when all private caches are accessed at the same time. This is particularly interesting because no contention appears even when coherence traffic is involved. However, as one would expect, contention seems to appear in shared resources such as L3 and memory. As explained above, the L3 cache contention is actually caused by parallel capacity misses caused by multiple threads sharing the overall L3 size.

More interestingly the ratio depends on the state of cache lines accessed: accessing modified cache lines (in L3) exhibits less performance than accessing clean lines. Also, we can see on Figure 4 that performance is more degraded by writing to exclusive cache lines than by writing to shared ones. We could not explain this behavior and it justifies further the idea to hide hardware complexity by using benchmarks: they capture such puzzling hardware behaviors more than abstracted analytical models, and we just use their outputs in our model.

V. Experiments

In this section we present several memory-bound applications that we modeled in order to predict their performance. Compute-bound applications are not considered because memory accesses are overlapped with computation and thus do not need to be optimized much.

One possible use of our model is to select the best working set size to achieve best performance. Another would be to select the minimal number of threads to use for a given computation in order to reach some level of performance. In order to illustrate these two approaches in the next sections, we will present comparison between our predictions and real applications. We only selected a few graphs in order to show interesting or unexpected results.

A. A Simple Code: Dot Product

The dot product computation is a simple code, consisting in the load of 2 different chunks. For our experiment, we assume all chunks are written beforehand by thread 0, therefore all chunks are in state $(M, \{0\})$. We choose to initialize vectors this way because it shows first, that the initialization phase can be critical for further performance, second because that is what many users would do in the first place. The performance prediction is compared to measurements of the MKL library dotproduct.

For both arrays read by dot product and for all of their chunks, the transition taken in Automaton 2 is a $\mathbf{L}_{\mathcal{T}}$ from state $(\mathbf{M}, \{0\})$. When there is more than one thread, this leads to state $(\mathbf{S}, \mathcal{T})$, while there is no state change with only thread 0. Since there is no chunk store in the dot product, the timing function is the sum of the timing functions for these transitions.

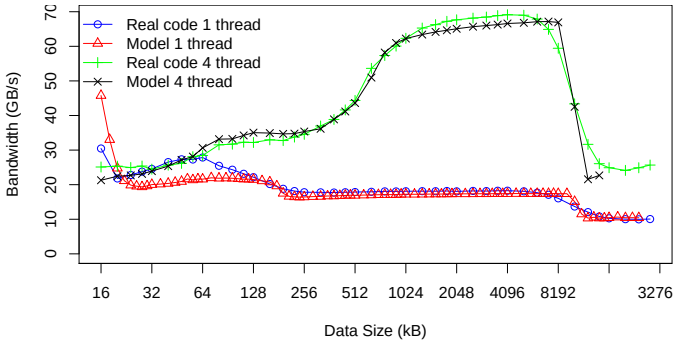

Figure 5: Dotproduct performance for 1 and 4 threads.

Figure 5 shows that our model is able to predict the behavior of dotproduct function calls from the MKL library.

The figures show that when only using one thread the dot product computation is faster when data set fits in the L1 cache while it is faster in L3 when using several threads. Again, this comes from coherence overhead: with a single thread, there is no coherence to maintain, thus we get better performance with faster memory. However with more threads, coherence gets involved due to the initialization of the vectors and performance is degraded. But when data only fits in the last level of cache, which is shared between all cores, then no cache coherence is required anymore, and code achieves better performance.

However, if we are careful with vector initialization (*i.e.* computing threads initialize the chunks they read), the dotproduct kernel can exhibit super-linear speedup as show on Figure 6 This super-linear speedup is due to

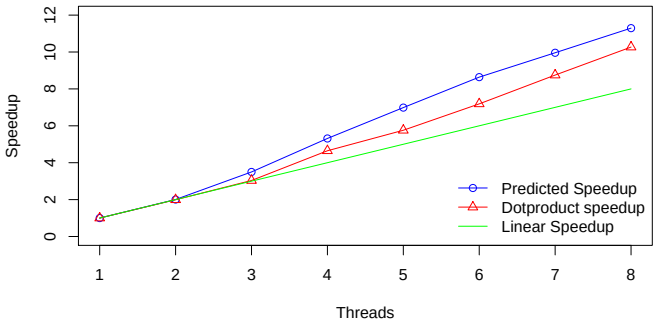

Figure 6: Dotproduct strong scalability on 512 KB vectors.

the size of the chunks accessed by threads. With a single thread, 2 512 KB-wide chunks are accessed and they only fit in the L3 cache. However with more threads the chunk size becomes smaller and they fit in lower cache levels, leading to better performance.

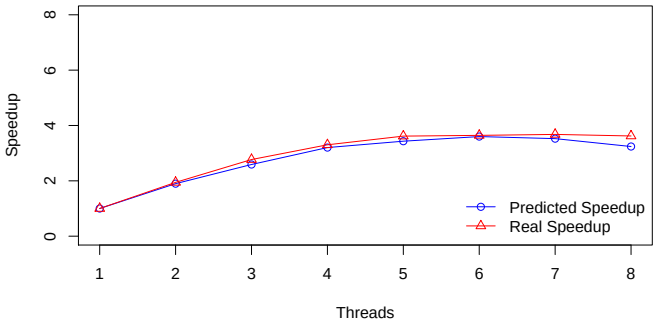

Figure 7: Dotproduct strong scalability on 32 MB vectors.

Yet, with very large chunks, the dotproduct kernel can have a poor speedup even with a careful data initialization as shown on Figure 7. It increases up to 4 with 6 threads and then stagnates. The predicted speed-up is close to the measured one. As seen in Section IV-B, parallel capacity misses appear when all threads use the same shared cache, leading to poor scalability. Even with 8 threads running this kernel, each of them still manipulates 8 MB which does not fit in local caches. This is an example showing that our model handles parallel capacity misses: when multiple threads compete for the memory blocks of a shared cache, our performance model is able to take into account the capacity misses that occur. Moreover, while this is not observed here, memory contention could be predicted likewise.

B. Comparing Iterative Methods

Iterative Krylov methods are crucial algorithmic patterns for many numerical simulations, in particular for solving Partial Differential Equations. They are all built

from the same building blocks (BLAS-1 and BLAS-2 functions) but exhibit different behaviors in term of convergence (application dependent) and time per iteration (architecture dependent). We compare in the following the performance prediction and measurement of one iteration of different Krylov methods, solving the equation $Ax = b$ where x is an unknown vector, b a constant vector and A a matrix. The methods considered are Conjugate Gradient Normal Residual (CGNR), Conjugate Residual Normal Error (CRNE) and Biconjugate Gradient Stabilized (BiCGSTAB) [11]. The code we use is adapted from a Lattice QCD simulation code [12] and uses OpenMP C code. The matrix A is a 8000 sparse matrix of complex in double precision, consisting in small blocks of dense matrices on the diagonal and subdiagonals. As we are focusing only on the time per iteration, the iterative methods are arbitrarily stopped after 200 iterations.

All these methods use different access patterns, different working sets per thread, and have possible coherency issues (with dot product in particular). The scalability of the time per iteration for BiCGSTAB method is shown in Figure 8. The performance increases up to a maximum

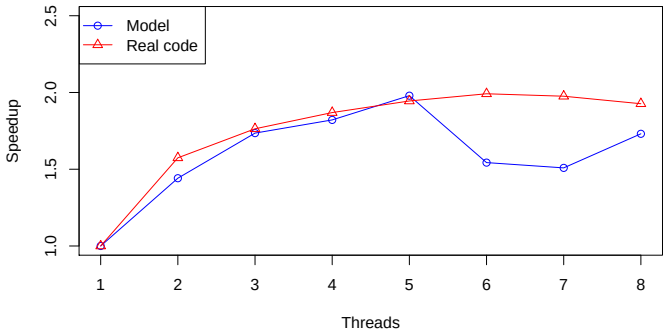

Figure 8: Speedup of the Bi-Conjugate gradient, Stabilized (BiCGSTAB).

of 5 threads according to our model (6 according to experimental results).

The timings of different methods are compared, for different thread numbers. Figure 9 shows that the time prediction is accurate enough to predict how performance changes when parallelism increases, and how different methods compare.

VI. Related Work

Several methods are commonly used to optimize software by observing and predicting performance. One is to simulate the full hardware, for instance with cycle accurate simulators [13], [14]. Such predictions are very precise and permit collection of large amount of performance metrics. However they are time consuming and require a deep knowledge of the hardware in order to implement all architecture features, including prefetchers or cache replacement policies, with enough precision to provide cycle accurate simulation. Developing such simulation software is a long process for each newly supported platform,

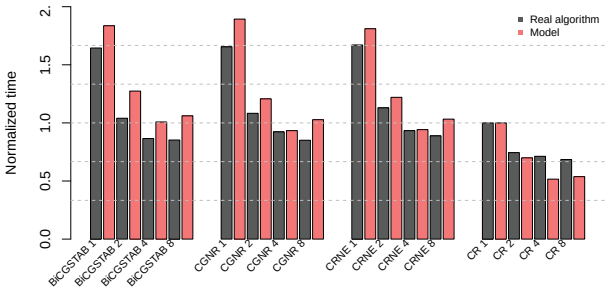

Figure 9: Performance for one iteration of different iterative methods. Performance figures are normalized, 1 corresponds to the time for the best sequential method. Two bars are given for Method n, corresponding to the normalized time when using n threads: in black, measured time, in red the predicted time.

and it highly depends on the hardly-available hardware documentation. However, such simulation is able to explain precisely what happens for a given code fragment. Our approach hides this complexity in the benchmarks and tries to remain portable by using a memory model that matches most widely-available modern processors and coherence protocols.

Profiling can also be used in order to record performance metrics for HPC applications tuning. Profiling has the same drawbacks as simulation since it slows down application performance. Tools such as Valgrind or Cachegrind [15] can present an overhead up to 100 times the normal program execution time. Our approach is only time consuming when running the benchmarks once for each platform. Again, these approaches are complementary to the technique we propose, since profiling and simulation are able to provide more details concerning the hardware mechanisms triggered during a code execution.

Hardware counters are tools for performance tuning. More elaborated tools have been developed in order to ease the use of hardware counters [16], [17]. The advantage of counters compared to simulators is that it is lightweight: There is no overhead aside from the library initialization. However counters are not enough to optimize software. Indeed once a bottleneck of the application is found (let say, too many TLB misses), one needs a way to link the information back to source code in order to tackle the problem. Also, as discussed in Section IV-B, the overhead of misses significantly varies with cache states.

Hackenberg *et al.* compared cache coherence of real world CPUs in [18]. They show that cache coherency and cache data states are to be taken into account when modeling memory hierarchy. Williams *et al.* proposed an ingenious work in modeling both memory and computation in order to predict best achievable performance of a given code depending on its arithmetic intensity[19]. Ilic *et al.* extended the model to support caches and data reuse [20]. Compute-bound applications are handled while we are not able to predict computation performance.

However, our model is able to predict in a better way applications with heavy coherence traffic. This also allows us to point out that bad performance of some applications can come from a huge overhead due to cache coherence. The references confirm the relevance of our approach for modeling memory access performance.

Our approach differs from all existing ones as the memory model is based on benchmarks. Benchmarks, especially the ones focusing on memory, have been developed in order to understand memory or application performance [21], [22]. They are a great way to understand architecture behavior, however they can not be directly used to optimize software. Once our model is built for a given architecture, we are able to predict both software scalability and achievable memory bandwidth. By understanding the memory model or predicted scalability, one can see if performance is limited by memory contention or because of a cache coherence unfriendly memory access pattern. Also, as our contribution is based on a model, there is no need to run the targeted application to predict its performance.

VII. Conclusion and Future Work

Code simulation and performance prediction become critical for performance analysis and software tuning. This paper presented an innovative model that predicts the performance of memory-bound OpenMP applications by composing the output of micro-benchmarks based on the state of data buffers in hardware caches. The model successfully predicts the execution time for several commonly-used application patterns, in particular those based on BLAS, and the effects of coherency and contention on performance. Additionally, our model incorporates the effects of compulsory and, to some extent, capacity misses. The systematic use of micro-benchmarks makes the model accurate without paying the cost to model complex hardware mechanisms such as prefetchers and cache coherency protocol implementations.

For future work, we plan to add support for multiple processor sockets. Loss of performance resulting from inter-socket interactions are ignored in our current model. While benchmarks are able to measure threads binded on different sockets, we need to plug them into the model, avoiding the combinatorial explosion coming from the exploration of the different bindings. Besides, we are thinking of adding automatic ways to detect coherence issues, for instance by defining some metrics based on the model and looking at applications feedback with hardware counters.

References

[1] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *Intl. Conf. on Parallel, Distributed and Network-Based Processing*. Pisa: IEEE, Feb. 2010, pp. 180–186.

[2] J. X. and X. Vera, "Efficient and Accurate Analytical Modeling of Whole-Program Data Cache Behavior," *Computers, IEEE Transactions on*, vol. 53, no. 5, pp. 547–566, 2004.

[3] C. Cascaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Intl. conf. on Supercomputing*. New York, NY, USA: ACM, 2003, pp. 150–159. [Online]. Available: http://doi.acm.org/10.1145/782814.782836

[4] D. Andrade, B. B. Fraguela, and R. Doallo, "Accurate prediction of the behavior of multithreaded applications in shared caches," *Parallel Computing*, vol. 39, no. 1, pp. 36–57, 2013.

[5] J. Lee, H. Wu, M. Ravichandran, and N. Clark, "Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications," in *Intl. Symp. on Computer architecture*. New York: ACM, 2010, pp. 270–279. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815996

[6] M. Papamarcos and J. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *SIGARCH Comput. Archit. News*, vol. 12, no. 3, pp. 348–354, Jan. 1984. [Online]. Available: http://doi.acm.org/10.1145/773453.808204

[7] E. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" *SIGPLAN Not.*, vol. 45, no. 5, pp. 203–212, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837853.1693482

[8] B. Putigny, "Mbench: memory benchmarking framework for multicores," https://github.com/bputigny/mbench.

[9] J. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pp. 19–25, Dec. 1995.

[10] A. Pesterev, N. Zeldovich, and R. T. Morris, "Locating Cache Performance Bottlenecks Using Data Profiling," in *European Conf. on Computer systems*. New York, NY, USA: ACM, 2010, pp. 335–348. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755947

[11] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, May 2007.

[12] D. Barthou, O. Brand-Foissac, R. Dolbeau, G. Grosdidier, C. Eisenbeis, M. Kruse, O. Pène, K. Petrov, and C. Tadonki, "Automated code generation for lattice quantum chromodynamics and beyond," *CoRR*, vol. abs/1401.2039, 2014.

[13] C. Hughes, V. Pai, P. Ranganathan, and S. Adve, "Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors," *Computer*, vol. 35, no. 2, pp. 40–49, 2002.

[14] R. Covington, S. Dwarkada, J. R. Jump, J. B. Sinclair, and S. Madala, "The Efficient Simulation of Parallel Computer Systems," in *Intl. J. in Comp. Simulation*, 1991, pp. 31–58.

[15] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," in *Workshop on Runtime Verification*, 2003.

[16] P. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A Portable Interface to Hardware Performance Counters," in *In Proc. of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[17] S. Jarp, R. Jurga, and A. Nowak, "Perfmon2: A leap forward in Performance Monitoring," *J.Phys.Conf.Ser.*, vol. 119, p. 042017, 2008.

[18] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems," in *Intl. Symp. on Microarchitecture*. New York: ACM, 2009, pp. 413–422. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669165

[19] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[20] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware Roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 99, no. RapidPosts, p. 1, 2013.

[21] J. Treibig, G. Hager, and G. Wellein, "Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering," in *Intl. conf. on Parallel processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 451–460. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36949-0_50

[22] J. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/